UNIVERSIDAD AUTONOMA DE COAHUILA.

FACULTAD DE SISTEMAS.



Algoritmos de ordenamiento y busqueda.

Longest Alternating Subarray problem.

Profesor:

Santiago Chío Benavides.

Alumno:

Alfonso Ramírez Cárdenas 13532045.

Fecha: viernes 24 de noviembre del 2023.

**EXPLICACIÓN DEL PROBLEMA.**

El Problema de la Subsecuencia Alternante Más Larga (LAS: Longest Alternating Subsequence) es un problema de optimización en el campo de la informática y la programación. Se trata de encontrar la longitud de la subsecuencia más larga en una secuencia dada, de tal manera que los elementos seleccionados en la subsecuencia tengan un orden estrictamente creciente y decreciente de manera alternante. En este contexto, una subsecuencia es una secuencia de elementos que mantiene el orden relativo de los elementos originales, pero no necesariamente de forma contigua.

Para entender mejor el problema, definamos algunos términos:

1. **Subsecuencia:** Una subsecuencia de una secuencia dada es un conjunto de elementos que mantienen el orden relativo de aparición, pero no necesariamente son contiguos. En otras palabras, puedes eliminar algunos elementos de la secuencia original sin cambiar el orden de los elementos restantes.
2. **Subsecuencia Alternante:** Una subsecuencia es alternante si los elementos seleccionados tienen un orden estrictamente creciente y decreciente de manera alternante. Por ejemplo, en la secuencia {1, 5, 2, 3, 4}, una subsecuencia alternante podría ser {1, 5, 2, 4}.
3. **Longitud de la Subsecuencia:** La longitud de una subsecuencia es simplemente el número de elementos que contiene.

El problema LAS se plantea de la siguiente manera: Dada una secuencia de elementos, ¿cuál es la longitud máxima de una subsecuencia alternante?

Este problema puede abordarse utilizando programación dinámica. La idea general es mantener dos matrices, una para la longitud de la subsecuencia alternante creciente hasta el momento y otra para la longitud de la subsecuencia alternante decreciente hasta el momento. Luego, para cada par de elementos en la secuencia, se actualizan estas matrices en función de las reglas del problema.

La solución óptima se encuentra tomando el máximo valor entre las matrices de longitud de subsecuencia creciente y decreciente en cada posición.

**SOLUCIONES DEL PROBLEMA.**

La solución al Problema de la Subsecuencia Alternante Más Larga implica el uso de programación dinámica para encontrar la longitud de la subsecuencia más larga que cumple con las condiciones establecidas (alternancia de crecimiento y decrecimiento).

**Paso 1: Definir el Problema**

Dado una secuencia de números, queremos encontrar la longitud de la subsecuencia alternante más larga. Una subsecuencia es alternante si los elementos seleccionados tienen un orden estrictamente creciente y decreciente de manera alternante.

**Paso 2: Formular una Estrategia**

La estrategia general es utilizar programación dinámica para construir dos matrices que representan la longitud de la subsecuencia alternante creciente y decreciente hasta cada posición en la secuencia. Luego, se toma el máximo valor en cada posición de ambas matrices para obtener la longitud de la subsecuencia alternante más larga.

**Paso 3: Desarrollar el Algoritmo**

El siguiente algoritmo resuelve este problema utilizando Programación dinámica. Alterna números positivos y negativos.

un main() {  
 *println*("Ingrese números enteros; positivos y negativos, separados por espacios para formar el arreglo:")  
 val input = *readLine*()  
 val arreglo = input?.*split*(" ")?.*mapNotNull* **{ it**.*toIntOrNull*() **}**?.*toIntArray*()  
  
 if (arreglo != null) {  
 val resultado = *encontrarSubsecuenciaAlternanteDinamica*(arreglo)  
 *println*("La longitud de la subsecuencia alternante más larga es: ${resultado.first}")  
 *println*("La subsecuencia alternante más larga es: ${resultado.second.*joinToString*(" ")}")  
 } else {  
 *println*("Error: Ingrese números enteros válidos.")  
 }  
}  
fun encontrarSubsecuenciaAlternanteDinamica(arreglo: IntArray): Pair<Int, List<Int>> {  
 val n = arreglo.size  
 val dp = IntArray(n) **{** 1 **}** val predecesor = IntArray(n) **{** -1 **}** for (i in 1 *until* n) {  
 for (j in 0 *until* i) {  
 if ((arreglo[i] > 0 && arreglo[j] < 0) || (arreglo[i] < 0 && arreglo[j] > 0)) {  
 if (dp[i] < dp[j] + 1) {  
 dp[i] = dp[j] + 1  
 predecesor[i] = j  
 }  
 }  
 }  
 }  
 var maxIndex = dp.*indexOf*(dp.*maxOrNull*() ?: 1)  
 val subsecuencia = *mutableListOf*<Int>()  
  
 while (maxIndex != -1) {  
 subsecuencia.add(arreglo[maxIndex])  
 maxIndex = predecesor[maxIndex]  
 }  
 return Pair(dp.*maxOrNull*() ?: 1, subsecuencia.*reversed*())  
}

**¿Como funciona la función encontrarSubsecuenciaAlternanteDinamica?**

1. **Parámetros de entrada:**
   * **arreglo**: Un arreglo de enteros que se pasa como entrada a la función.
2. **Inicialización de variables:**
   * **n**: La longitud del arreglo, es decir, el número de elementos en el arreglo.
   * **dp**: Un arreglo de longitud **n** que almacena la longitud de la subsecuencia alternante más larga que termina en cada posición del arreglo.
   * **predecesor**: Un arreglo de longitud **n** que almacena el índice del elemento anterior en la subsecuencia alternante más larga que termina en cada posición.
3. **Bucle principal:**
   * La función utiliza un bucle anidado para comparar cada par de elementos en el arreglo.
   * Si la condición **(arreglo[i] > 0 && arreglo[j] < 0) || (arreglo[i] < 0 && arreglo[j] > 0)** se cumple, significa que los elementos en las posiciones **i** y **j** tienen signos alternantes.
   * Se compara la longitud de la subsecuencia alternante más larga que termina en la posición **i** con la longitud de la subsecuencia que termina en la posición **j** más 1.
   * Si es mayor, se actualiza la longitud y se guarda el índice del predecesor.
4. **Construcción de la subsecuencia:**
   * Se encuentra el índice del máximo valor en el arreglo **dp**, que representa la longitud de la subsecuencia alternante más larga.
   * Se construye la subsecuencia inversa utilizando el arreglo **predecesor** y se almacena en la lista **subsecuencia**.
5. **Resultado:**
   * La función devuelve un par que consiste en la longitud de la subsecuencia alternante más larga y la subsecuencia en sí misma.

En resumen, la función utiliza programación dinámica para encontrar la longitud de la subsecuencia alternante más larga y luego reconstruye la subsecuencia a partir de la información almacenada en el arreglo **predecesor**.

**Paso 4: Analizar la Complejidad**

1. **Inicialización de arreglos:** La creación de los arreglos **dp** y **predecesor** tiene una complejidad de O(n), ya que ambos tienen un tamaño proporcional al tamaño del arreglo de entrada.
2. **Bucle anidado:** El bucle anidado tiene dos niveles. El primer bucle **for** itera a través de los elementos del arreglo (**i**), y el segundo bucle **for** itera a través de los elementos anteriores a **i** (**j**). Dentro del bucle anidado, hay operaciones de comparación y asignación, pero no hay bucles adicionales. Por lo tanto, el bucle anidado tiene una complejidad de O(n^2).
3. **Bucle While:** Después de los bucles anidados, hay un bucle **while** que sigue los predecesores para construir la subsecuencia. Este bucle tiene una complejidad de O(n), ya que, en el peor caso, podría recorrer todos los elementos del arreglo una vez.

Por lo tanto, la complejidad total del algoritmo es O(n^2 + n), pero en el análisis asintótico, el término dominante es O(n^2). Por lo tanto, la complejidad final del algoritmo es O(n^2).

**SOLUCIÓN UTILIZANDO PROGRAMACIÓN LINEAL.**

fun main() {  
 *println*("Ingrese números enteros separados por espacios para formar el arreglo:")  
 val input = *readLine*()  
 val arreglo = input?.*split*(" ")?.*mapNotNull* **{ it**.*toIntOrNull*() **}**?.*toIntArray*()  
  
 if (arreglo != null) {  
 val resultado = *encontrarSubsecuenciaAlternanteLineal*(arreglo)  
 val longitudSubsecuencia = resultado.first  
 val subsecuencia = resultado.second  
  
 *println*("La longitud de la subsecuencia alternante más larga es: $longitudSubsecuencia")  
 *println*("La subsecuencia alternante más larga es: $subsecuencia")  
 } else {  
 *println*("Error: Ingrese números enteros válidos.")  
 }  
}  
  
fun encontrarSubsecuenciaAlternanteLineal(arreglo: IntArray): Pair<Int, List<Int>> {  
 if (arreglo.*isEmpty*()) return Pair(0, *emptyList*())  
  
 var longitudMaxima = 1  
 var longitudActual = 1  
 var inicioSubsecuencia = 0  
 var inicioMaximo = 0  
  
 for (i in 1 *until* arreglo.size) {  
 if ((arreglo[i] > 0 && arreglo[i - 1] < 0) || (arreglo[i] < 0 && arreglo[i - 1] > 0)) {  
 longitudActual++  
 if (longitudActual > longitudMaxima) {  
 longitudMaxima = longitudActual  
 inicioMaximo = inicioSubsecuencia  
 }  
 } else {  
 longitudActual = 1  
 inicioSubsecuencia = i  
 }  
 }  
  
 val subsecuencia = arreglo.*copyOfRange*(inicioMaximo, inicioMaximo + longitudMaxima).*toList*()  
  
 return Pair(longitudMaxima, subsecuencia)  
}

**¿Como funciona la función encontrarSubsecuenciaAlternanteDinamica?**

1. **Manejo de Caso Base:**
   * Si el arreglo de entrada está vacío, la función devuelve una pareja **(0, emptyList())**, lo que significa que la longitud de la subsecuencia alternante más larga es cero y la subsecuencia en sí es una lista vacía.
2. **Inicialización de Variables:**
   * **longitudMaxima**: Se inicializa con 1, ya que una subsecuencia alternante debe tener al menos un elemento.
   * **longitudActual**: Se inicializa con 1, ya que en este punto se asume que hay una subsecuencia de longitud 1.
   * **inicioSubsecuencia**: Se inicializa con 0, indicando el índice donde comienza la subsecuencia actual.
   * **inicioMaximo**: Se inicializa con 0, este será actualizado cuando se encuentre una subsecuencia alternante más larga.
3. **Iteración sobre el Arreglo:**
   * Se itera sobre el arreglo comenzando desde el segundo elemento (índice 1) hasta el final.
   * Para cada par de elementos adyacentes, se verifica si uno es positivo y el otro es negativo. Si es así, se considera que se ha encontrado un elemento alternante en la subsecuencia y se incrementa **longitudActual**.
   * Si **longitudActual** supera a **longitudMaxima**, se actualiza **longitudMaxima** y **inicioMaximo**.
   * Si los elementos adyacentes no son alternantes, se reinicia **longitudActual** a 1 y se actualiza **inicioSubsecuencia** al índice actual.
4. **Construcción de la Subsecuencia Resultante:**
   * Después de la iteración, se utiliza **inicioMaximo** y **longitudMaxima** para extraer la subsecuencia alternante más larga del arreglo original.
   * Se utiliza **copyOfRange** para obtener una porción del arreglo original y luego se convierte en una lista.
5. **Retorno del Resultado:**
   * La función devuelve una pareja que contiene la longitud de la subsecuencia alternante más larga (**longitudMaxima**) y la subsecuencia en sí (**subsecuencia**).

En resumen, la función utiliza un enfoque lineal para encontrar la longitud y la posición de la subsecuencia alternante más larga en el arreglo dado.

**Complejidad (Big O):** O(n)

**Explicación:**

1. El código utiliza un bucle **for** para iterar a través del arreglo una vez.
2. En cada iteración, se realiza una comparación simple **(arreglo[i] > 0 && arreglo[i - 1] < 0) || (arreglo[i] < 0 && arreglo[i - 1] > 0)** para determinar si los elementos adyacentes tienen signos alternantes.
3. Se actualizan las variables **longitudActual**, **inicioSubsecuencia**, **longitudMaxima**, y **inicioMaximo** en consecuencia.
4. Al final, se copia la subsecuencia más larga encontrada en una lista.

Dado que cada elemento del arreglo se procesa una vez y no hay bucles anidados, la complejidad temporal es lineal con respecto al tamaño del arreglo, es decir, O(n), donde n es la longitud del arreglo.